Exploring the Potentials of Parallel Garbage Collection in SSDs for Enterprise Storage Systems

Narges Shahidi† Mohammad Arjomand† Myoungsoo Jung†
Mahmut T. Kandemir† Chita R. Das† Anand Sivasubramaniam†

†Pennsylvania State University
{nxs314,mxa51}@psu.edu {kandemir,das,anand}@cse.psu.edu
‡School of Integrated Technology, Yonsei Institute Convergence Technology, Yonsei University
mj@camelab.org

Abstract—In the last decade, NAND flash-based SSDs have been widely adopted for high-end enterprise systems in an attempt to provide a high-performance and reliable storage. However, inferior performance is frequently attained mainly due to the need for Garbage Collection (GC). GC in flash memory is the process of identifying and clearing the blocks of unneeded data to create space for the new data to be allocated. GC is a high-latency operation and once it is scheduled to run on a block of a plane in a flash chip (each flash chip consists of multiple planes), it can increase latency for later arriving I/O requests to the same plane. Apart from that, the consequent high latency also keep other planes of the same chip, that are not involved in this GC, idle for a long time. We show that for the baseline SSD with modern FTL, GC considerably reduces the plane-level parallelism, causing significant performance degradation. There are several circuit-level constraints that make it difficult to allow subsequent I/O operations and/or GCs to be served concurrently from the same chip, but different planes, during the long latency GC. This paper proposes a novel GC strategy, called Parallel GC (PaGC), whose goal is to proactively run GC on the remaining planes of a flash chip whenever any of its planes needs to execute on-demand GC. The resulting PaGC system boosts the response time of I/O requests by up to 45% (32% on average) for different GC settings and across a wide spectrum of enterprise I/O workloads.

I. INTRODUCTION

Large-scale and high-end enterprise systems typically employ Solid-State Disks (SSDs) to achieve their performance and reliability goals [1]–[5]. SSDs in modern enterprise systems usually have tens to hundreds of NAND-flash memory chips and multiple processing cores that are interconnected by multiple I/O channels to form a single storage [6]. A single flash chip nowadays consists of multiple dies and multiple planes within each die. Thanks to these intra-chip and inter-chip architecture, one can expect superior performance and bandwidth improvement in modern enterprise SSDs compared to conventional storage media. However, as expected, performance characteristics of these SSDs vary based on how well these parallel resources, especially hundreds of flash dies and planes, are utilized [7], [8]. In order to efficiently manage these resources, Flash Translation Layer (FTL) for SSDs have been undergoing severe design changes. However, there are a number of critical performance bottlenecks that cannot be hidden, among which Garbage Collection (GC) might be one of the most important issues [9]–[11].

Writes on SSDs are made in size of flash pages (typically 2KB–8KB), and these writes are made in an out-of-place fashion – i.e., rather than modifying existing data, the previous flash page is invalidated and the new copy of data is written into another free page. When the prepared free pages are used up, the FTL reclaim block(s) of pages as victims. Before reclaiming a block, if the block has valid pages, the FTL needs to reallocate them to newly-allocated blocks from the free pool, followed by erasing the old block(s). This block reclaiming process is referred to as garbage collection. The page movements (read/writes) necessary to perform this remapping of valid data as well as erase operations make GC a time-consuming activity, which largely affects the SSD throughput and the worst-case response time of I/O requests. As expected, our experiments show that GC lead to frequent drops in system performance, reducing I/O requests’ response time by up to 400% in the baseline high-end SSDs for enterprise workloads.

GC implementations have been extensively studied in the context of various FTL, but to date almost all works have focused on reducing GC execution time [12]–[14], investigating various victim block selection algorithm [15]–[17], or hiding GC latency by performing it at the time that SSD is idle [18]. This paper, however, targets a different class of GC optimization by analyzing the impact of GC on the flash utilization at the finest granularity in modern flash chips (i.e., plane-level utilization), and proposes a novel GC mechanism to boost SSD utilization and performance.

Modern flash chips are usually made up of multiple independent dies and multiple planes inside a die. While dies almost work independently, I/O operations on multiple planes are allowed but have certain restrictions: they should be of the same type (read, write or erase) and the target pages should have same address offset. In current flash chips, this type of accesses are handled by means of multi-plane commands. In this organization, when the FTL invokes GC on one of the planes, because of restrictions of the internal circuitry of the flash chips, the other plane cannot service I/O requests and remain idle. To the best of our knowledge, this type of plane-level under-utilization due to GC has not been studied in literature, and is the target of our optimization in this paper.

We propose Parallel GC (PaGC), a novel GC strategy to enhance the plane-level utilization during GC and consequently
reduce the impact of GC on the overall SSD performance. The primary idea behind PaGC is to perform multiple GCs on separate planes at the same time. In other words, when FTL invokes GC on a plane, it also runs another GC on the other plane (or paired planes), and tries to simultaneously progress both GCs by managing page movements using multi-plane command. Consequently, with PaGC, we perform an on-demand GC on one of the planes while the other plane is performing early GC. Our experimental results show that the write traffic patterns on the paired planes are quite similar, hence early GC does not impose an extra cost in terms of the increased number of block erasures in the paired plane(s).

Our extensive experimental study with different GC settings and for a variety of I/O workloads demonstrates that PaGC improves SSD performance in two ways: (1) by performing early GC on the non-demanding plane, it enhances the plane-level utilization and reduces the flash chip/die stall-time due to frequent GC invocations; and (2) by properly managing page movements within each plane using multi-plane commands, PaGC is able to perform a majority of data movements in a completely parallel fashion and with the latency of a basic data movement (one read and one write). Hence, the extra latency overhead incurred by PaGC is very negligible. PaGC is simple to implement and imposes only marginal additions to the FTL software. This paper makes the following contributions:

- We quantitatively analyze the plane-level utilization of an SSD when one of the planes in a die is performing GC. Our analysis show that, conflicts due to GC on non-native I/O request (i.e., I/O requests that are not targeting the plane under GC) is considerable (up to 40%), degrading request response time significantly. This class of SSD under-utilization has not been studied before and is the main target of our proposed optimization in this paper.

- We propose PaGC, a novel GC strategy targeting SSDs with modern flash memories, that aims to increase plane-level parallelism during GC. When the FTL invokes PaGC on a set of paired planes, it reclaims blocks on different planes, and reduces the cost of page movements by performing most of them in parallel using the multi-plane command.

- We experimentally show that when PaGC is executed blindly by the FTL (i.e., performing GC on all planes of a die whenever one of the planes needs GC), the SSD performance degrades for some workloads, because of the extra cost imposed by page movements that have to be serialized. To reduce this overhead, we propose a threshold-based PaGC, a scheme that invokes GC only when the number of free pages on the planes not-involved in GC is more than a threshold.

- By examining the SSD performance for a variety of I/O workloads under different page-level allocations and GC settings, our experimental results show that PaGC can improve the request response time by up to 45% (32% on average).

II. RELATED WORK

Garbage Collection cause variability in performance and increase worst case response time. To address this issue, Lee et.al in [10] propose Preemptive Garbage Collection (PGC) that preempt the on-going GC process and serves incoming requests over regular GC activity. The paper split GC operations into distinct operations and insert incoming requests in between them. The paper also pipelines incoming requests with an operation of GC. Similarly, [18] targets violation of quality of service due to GC and propose a host interface I/O scheduler that is both GC-aware and QoS-aware. This scheduler re-distributes GC overheads across non-critical requests to reduce worst case latency in NAND Flash based SSDs. In order to mitigate the performance variability, our approach reduces the time that a die spends on GC (reduce GC invocation frequency) and perform GC more efficiently by making it in parallel for planes of a die.

Another group of prior works leverage advanced commands in order to reduce garbage collection overhead. Tavakkol et.al. in [19] propose a LPA to PPA mapping which exploit the plane-level parallelism. This paper targets a long time benefit of improving plane-level parallelism to mitigate the negative impact of GC. The paper leverages multi-plane and copy-back operations to reduce garbage collection overhead.

Another prior work in this group is [20] that proposes DLOOP which allocates logs onto the same plane where their associated original data resides. This way, copying valid pages triggered by garbage collection can be carried out by copy-back operations without occupying the external IO bus. This paper optimize page-mapping FTL to distribute request across all planes evenly and exploit plane-level parallelism. In our work, in contrast, we do not change mapping, however, we leverage multi-plane operation on two or more planes of a die to improve plane utilization and GC efficiency.

III. BACKGROUND

A. SSD and NAND flash chip organization

Enterprise SSDs usually have a multi-chip multi-channel organization as illustrated in Figure 1. This is composed of four components: 1) Host Interface (HI) provides communication with the host and is responsible for queuing I/O requests and sending responses back to the host; 2) The processor is responsible for processing I/O requests and executing FTL firmware; 3) Multiple Flash Chip Controllers (FCC) are responsible for command and data transfer between the processor and flash chips; and 4) A set of flash chips provide storage and are connected to FCC through multiple I/O channels.

Figure 1 also shows a NAND flash chip which is internally composed of two or more dies and each die has multiple planes. Each plane contains thousands of blocks and one or more data/cache registers used as buffers. A flash block
typically consists of 64, 128 or 256 pages. The size of a flash page is 2KB, 4KB, 8KB, and 16KB in recent products [21–25]. The flash memory has two unique characteristics, namely write-after-erase and erase cycle. A write operation can only change the value of a cell from ‘1’ to ‘0’. Once it is written, it must be erased (all cells are set to ‘1’), before the next write is performed on the same page. While the read and write operations are performed at the granularity of a page, the erase operation is performed at a block-level. To postpone erase operations, on a page update, the physical page containing old data is marked invalid and new data is written to an arbitrary free (clean) physical page turning its state to valid. This is known as the out-of-place update policy. Another characteristic of flash memory is that each block has a limited number of erase cycles before it is worn out. After wearing out, a block can no longer store any data. A typical NAND flash has erase-cycle limit of 10K–100K [26].

B. Advanced Commands in Modern NAND flash Memories

In addition to basic commands (read, write and erase), NAND flash memories support a set of advanced commands. These advanced commands are copy-back, multi-plane, and multi-die interleave, which are extensions of basic commands with some usage restrictions [21–25], [27].

Copy-back. This command moves data of a page to another page in the same plane without occupying the internal and external I/O buses of the chip. Although in early products with copy-back support [28], [29], it was assumed that the address of the source page and destination page must be both odd or both even, this restriction has been relaxed in current products, and is not considered in our proposed design.

Multi-plane. This command enables multiple reads, programs and erases in multiple planes of the same die in a completely parallel fashion. In modern flash chips, a multi-plane command usually targets two planes or, in its general form, $2^N$ planes. The pages targeted by a multi-plane command must have the same chip, die, plane and page address, i.e., only the block number can be different for different pages. The multi-plane commands can significantly boost the flash performance by providing a high degree of parallelism within a chip [30–32], with the cost of only one basic operation.

Multi-die Interleave. This command executes several page reads, page writes, block erases, and two other types of advanced commands (copy-back and multi-plane) in different dies on the same chip simultaneously. Since dies are independent, there is no restriction for the interleave command.

C. FTL Functionalities

The FTL implements the core algorithms required for managing resources inside SSD and emulates the block-device interface like HDDs. Below, we go over the major functionalities of an FTL.

Data mapping. Due to out-of-place update in flash memories, the FTL must implement a mechanism to enable address redirection from a logical page address to physical addresses in an SSD. Prior works have extensively studied different strategies [12] (page-level mapping, block-level mapping and hybrid mapping) and discuss their respective advantages and disadvantages. In this paper, we restrict our attention to pure page mapping, where each logical page can be freely assigned to any logical page within SSD address space [12]. This page mapping scheme gives better performance than its competitors, but requires a large table for address redirection, which should be kept in a DRAM chip near to the controller. To reduce DRAM capacity, mapping algorithms like DFTL [33] offer feasible implementations of pure page mapping.

Wear-leveling. Due to the non-uniform distribution of writes, the update frequencies of physical pages are not identical. This causes the erase count of different blocks vary drastically, resulting in fast wear-out of some blocks while others are far from their endurance limit. To prolong SSD’s lifespan, the FTL uses a set of wear-leveling policies, such as endurance-aware page-level mapping [31] or exchanging data between highly-written and less-written blocks [13], [14].

Over-provisioning and GC. The overall SSD capacity is usually higher than the capacity visible to the user. The invisible capacity, which is around 7–25% of the total storage capacity, is known as the over-provisioning space, and managed by FTL. There are two main reasons for having such an over-provisioning space. First, the FTL uses this space for bad-block management; Second, the FTL uses this space to manage high write traffic. In order to provide good performance particularly in applications with small random writes, the flash controller manages the write operations in a log structure. This log structure can be accommodated in the over-provisioning space. Once the available capacity of the over-provisioning space has reached a pre-specified threshold, the FTL needs to free up more blocks in order to have a sufficient pool of free flash blocks, i.e. the GC process with details in the next section.

Each GC execution has three steps. First, a GCselect function chooses one or more candidate blocks. Second, the valid pages of the selected candidate block(s) are moved into a clean block(s) and their corresponding entries in the mapping table are updated. Third, the selected candidate blocks are erased and added to the free block pool. It is clear that the GC process, in terms of which and how, is a key determinant to the performance of an SSD. We describe different trade-offs in the following.

D. Which Blocks are Selected by GCselect?

Ideally, GC has to provide a large amount of free pages at low cost. The main performance cost in GC is due to the movement of valid pages between blocks. This cost can be minimized by selecting candidate blocks with maximum amount of invalid pages; this way, copying the valid pages incurs minimum additional writes while more free pages will be available (or lower GC rate). On the other hand, the GCselect function needs to be careful about the lifetime of blocks and try to uniformly distribute erases over all chips/blocks (better wear-leveling). Many GCselect algorithms are already proposed considering one or several of the above
cost-efficiency metrics. We describe three important GCselect algorithms that are later used for evaluation:

- **GREEDY** [17]: It selects a block with the least number of valid pages. This algorithm is expected to reduce the movement cost and GC rate. Its potential drawbacks are poor wear-leveling and high latency.
- **RGA (d-choice)** [15]: It chooses a random set of $d$ blocks and selects one with the fewest number of valid pages within them. RGA trades off wear-leveling for movement cost.
- **RANDOM** [16]: It selects the victim block randomly with no particular concern regarding the valid/invalid pages. A variation of this algorithm, RANDOM+, repeats the random function until finding a block with at least one invalid page.

E. How are Data Movements Performed?

Copying valid data pages of a block to another during GC is a costly process. If GC is executed globally (within all SSD address space), the pages read are brought out of one chip, and based on the mapping decision, they could be later written to (possibly another) flash chip. To remove this transfer cost, some FTLs perform GC locally, i.e., pages are moved from one block to another in the same chip, helping to reduce potential conflicts with other requests on shared channels and chips. To this purpose each plane has to have an active block with a pointer to the page which is the target of the next write operation (called “write point”). Once the active block was full, another clean block is selected from the same plane as active block and the write operations continue with the newly selected block.

We call this strategy, implemented in the conventional FTLs, as “serial GC”, because when GC is initiated by the controller, only one plane is involved with GC and data movement, and keep the other planes, not involved in GC, idle (since the address and controller registers are shared among all planes in a die). When using the serial GC, the GCselect is allowed to select candidate blocks from the blocks in the same plane, and thus the time complexity is not an issue for GREEDY algorithm, as a plane typically has thousands of blocks.

F. Performance Overheads due to GC

GC can be a serious impediment to SSD performance because during the movement of valid pages, the flash chip cannot respond to normal I/O operations, increasing their response time. More precisely, the GC affects I/O performance of an application in three ways:

- **Imposing write cliff**: At the time of GC, the number of unwanted writes (due to moving valid pages) increases sharply. This phenomenon is known as write cliff that causes sharp drops in the SSD throughput [7]. One way to alleviate the problem of write cliff is to perform GC in the background, when the SSD is idle. Despite its potential benefits, background GC is rarely used in modern SSDs, due to two main issues. First, it is always desired to reduce the number of GC invocations even if they are going to be executed in the background. This is more important in flash memories in sub-20nm technologies which have very limited cell-endurance. Consequently, many flash firmwares postpone GCs as much as they can, which leads to on-demand GC rather than background GC. Second, the background GC consumes power while the SSD is in the idle state, and it is difficult to preserve data consistency in the case of power outage. Most SSDs put a maximum bound on power usage when the device is in the idle mode. Therefore, the background GC is inherently limited in page reclaiming during the idle periods. Consequently, most vendors opt to perform on-demand GC rather than background GC.

- **Causing variations in response time**: Once the GC is invoked, all requests destined for the target plane and its paired planes must wait till the controller completes the GC. This resource conflict increases the response time of the requests arriving during the GC period. Once the garbage collection is done, the FTL resumes servicing queued requests. Thus, GC incurs fluctuations in request response time, which is not acceptable from a QoS point of view. Two parameters of GC plays a role in this response time variation: (1) GC delay and (2) GC rate. The former is the period during which the plane is not available for I/O request service; contributing to an increase in the response time of incoming requests. The latter, GR rate, reduces the system robustness as it creates more frequent variations in response time.

- **Lowering plane-level utilization**: When a GC is operating on blocks at a target plane, none of the planes that belong to the same die is available for the entire execution time of GC. In other words, from a request point of view, the GC rate is simply the sum of the GC rates of all planes paired with each other. This phenomenon roughly doubles the average waiting time for an I/O request to start its service since the requests not only have to wait while the target plane is busy performing GC, but also when the other planes in the same die are doing GC. The serial GC therefore leads to an underutilization of the parallel resources inside a flash chip. Figure 2 pictorially illustrates this underutilization. Without loss of generality, we assume each die has two planes. It assumes that a die has two planes, and Plane1 starts GC at time $t_1$ and ends at time $t_2$. During this time, Plane2 is idle (unable to service any write request due to the conflicts on command/address registers), and hence, it is underutilized for the $t_2-t_1$ time units. A similar underutilization scenario happens for Plane1 when GC is performing on Plane2; in this example, it starts at time $t_3$ and ends at $t_4$. We will discuss this plane-level underutilization for real workloads in the next section.

IV. MOTIVATIONAL EXAMPLE FOR LOW PLANE-LEVEL UTILIZATION DURING GC

Based on the sequence of transactions happening during serial GC on a single plane, the total response time of a request can be broken into five components:

- **Service time (ST)**: This component represents the actual service time of the request, i.e., the difference between the
time that the flash command is issued and the time that the flash memory finishes servicing the request.

- **Conflict with non-GC operations (CnGC):** This is the time period where the request is waiting in the queue because its target chip is busy with servicing other requests or the shared channel is allocated to the requests of the other chips.
- **Conflict with GC on the same plane (CsGC):** This refers to the time period that the request is waiting in the queue while its target plane is busy with GC.
- **Conflict with GC on the other planes (CoGC):** This accounts for the time period during which the request is waiting in the queue because of the GC activities on other planes in the same die.
- **Late conflict with non-GC operations (LC):** Consider a scenario where two requests have resource conflict (on shared channel or shared chip), but their arrival times are far away. Although these requests would not normally conflict when no GC is taking place, the request arrived first may have to wait and is likely to conflict with the second when the GC is completed. In other words, the different requests may conflict with each other, not because they arrive at the same time, but because of the long latency of the GC.

Figure 3 shows the breakdown of request response time for typical I/O workloads of different categories: streaming, fileserver, workstation, and database. The configuration setup is the same as Section VI; the only difference is that we used a large NCQ for the SSD. This helps avoiding requests to stall at the host, so that we can account for all types of conflicts. To understand how much GC really affects the request service time, Figure 4 assumes that GC latency is zero (i.e., no data movement and erase takes any time), and shows the request response time, normalized to the case where GC latency is accurately modeled. One can easily observe that GC is a major contributor to the overall response time of a request, either directly or indirectly. In the streaming workload, more than 50% of request response time is because of the service time (ST) and non-GC resource conflict (CnGC). In comparison, the conflicts due to GC (CsGC and CoGC) contribute to less than 10%. As shown in Figure 4, if all conflicts due to GC could be removed (ideal case), the request response time can be improved by 40%, on average, for this workload. The fileserver workload shows no GC conflict at its early execution period, but once GC comes into play, we witness dramatic resource conflicts among different requests. For this application, we see a very high spatial conflict (CoGC) but low late conflict (LC). Again, one can see that, in the ideal case (i.e., with GC latency set to zero), the request response time reduces as much as 70% for this workload. For the workstation workload, we see that the request response time of a request largely suffers from the resource conflicts with non-GC operations (CnGC). Even though GC is frequently required at the final phases of execution, we see that, in the ideal case, the request response time gets about 20% improvement. The database workload shows a constant behavior over the time,
V. THE PROPOSED GC STRATEGY: PARALLEL GC (PaGC)

Figure 5 illustrates how our proposed PaGC works for the example in Figure 2 – The main idea is that, when the FTL decides running GC in one of the planes, it does not let the other plane remain idle, as in the case of serial GC; instead, the FTL tries to make the other plane busy by invoking another GC over it. The primary goal behind PaGC is to free up more clean pages at each GC invocations, in an attempt to (1) reduce the stall time of chip and thus enhance response time variations caused by GCs, and (2) improve plane-level utilization.

The main knobs that enable PaGC are two sets of advanced commands (multi-plane and copyback); in Section V-A, we will describe how PaGC carefully utilizes these two classes of commands. Ideally, PaGC should not take a longer time than the serial GC (i.e., our baseline) at any instance of GC invocation. For example, in Figure 5, executing GC on Plane2 when it was idle during normal GC on Plane1 does not increase idle time of Plane1 more than baseline (i.e., t3-t2). This is not always the case – for instance, due to limitations imposed by the GCselect algorithm as well as the restrictions of multi-plane and copyback commands, if we are going to have PaGC at time t4 (i.e., at this time Plane2 starts GC and PaGC tries to parallelize Plane1 with it), the die stall time increases because the GC on Plane1 takes longer than t5-t4. In Section V-B, we will explain when PaGC is beneficial and discuss our design decisions.

A. How does FTL Perform PaGC?

When the FTL decides to perform PaGC, it initially selects separate candidate blocks at each plane, and then, tries to move the valid pages of the selected candidate blocks within each plane in parallel. Lastly, FTL sends commands to erase the candidate blocks in parallel. While erasing the blocks in different planes can be parallelized by employing the multi-plane erase command, having data movements at two planes in a completely parallel fashion is challenging.

Parallel data movement of valid pages in different planes requires reading the valid pages by a multi-plane read command (brining them into the cache register of planes but not transferring them out the chip), and writing them back to the corresponding planes by a multi-plane write command. The FTL must be lucky if it can perform each and every movement of valid pages completely parallel because the valid pages of candidate blocks of two planes are not always at the same offset, i.e., a requirement for a multi-plane read operation. Moreover, there is no guarantee that the write points of the active blocks in each plane are pointing to the same page offset, which is a requirement for a multi-plane write operation. Figure 6 shows different possible scenarios happening during parallel data movement by an example (assuming 8 pages per plane). Let us assume that the FTL selects Block A and Block B as candidate blocks for GC at Plane 1 and Plane 2, respectively. Block A has four valid pages at the offsets of 1, 2 and 3 and 4, and Block B has three valid pages at the offsets of 1, 4 and 5. For Plane 1 and Plane 2, the write points are at the offsets 3 and 5 of their active blocks (i.e. Block Act-1 for Plane 1 and Block Act-2 for Plane 2), respectively. Figure 6 shows that, in the first and second steps, the pages at offset 2 and 3 of Block A are read in serial and written in serial; in step 3, the write points are aligned in both planes, thus the pages at the offset 1 are read in parallel and written in parallel; similarly, in step 4, pages at the offset 4 of two candidate blocks are read in parallel and written in parallel; and lastly, the page 5 of Block B is serially read and serially written.

Clearly, serial read is inevitable when the valid page of one plane corresponds to an invalid page of the other (pages 2 and 3 of Block A and page 5 of Block B in the above example). However, the serial write can still be avoided. A solution is to invalidate the pages in one of the active blocks and make their write points aligned, ready for multi-plane
write. For the example in Figure 6, by invalidating 2 pages in the active block of Plane 1 (Block Act-1), the write points get aligned; enabling multi-write operation for three write-backs (one for pages 1 of two blocks, one for pages 4, and one for page 2 of Block A and page 5 of Block B). This is a reasonable solution, if the difference between of the page offsets of the write points of two planes is not too much.

Alternatively, the FTL may keep two active blocks at each plane. The first active block functions the same as in the case of conventional flash memories, and its write point can move up with no restriction (unaligned active block). The second active block, on the other hand, is customized for PaGC (aligned active block) – the FTL always keeps the write points of the aligned active blocks of the paired planes at the same offset. This enables us employ multi-plane write for most of the write-backs (they may be already read with multi-read or basic read command) without wasting any free pages. Here, fragmentation due to having two write points is not an issue and is limited only to the size of one block, i.e., 64 or 128 pages. The FTL cannot perform multi-write in only one situation, that is when one of the candidate blocks has more valid pages than its counterpart (this happens for one page of Block A in the above example). In this situation, the extra valid pages are read serially and written-back serially in the unaligned active block of the corresponding plane.

The data movements in the PaGC can be classified as:

1) **Parallel Read & Parallel Write (PaRD-PaWR):** Valid pages are at the same offset of the candidate blocks. We use multi-plane command to read them and multi-plane command to write them back in aligned active blocks. The cost is that of one read and one write operations.

2) **Serial Read & Parallel Write (SrRD-PaWR):** The pages are not at the same offset in the candidate blocks. They are read by one serial read for each page, followed by a multi-plane operation to write them back in the aligned active blocks of the associated planes. The cost of two-page movement is two serial reads and one multi-plane write.

3) **Single Read & Single Write (SrRD-SrWR):** One plane has more valid pages than the other. These extra valid pages have to be moved serially, i.e., one serial read and one serial write at the unaligned active block for each page.

To perform data movements in PaGC as fast as possible, the FTL first moves the valid pages that belong to the first category (i.e., PaRD-PaWR). It then selects the valid pages of different candidate blocks and pairs them for data movement according to the second category (i.e., SrRD-PaWR). Lastly, the remaining pages are moved according to the third category (i.e., SrRD-SrWR). Let us assume that \(K\) and \(L\) are the number of valid pages in two candidate blocks of different planes \((K > L)\). If \(K_a\) is the number of pages in the category PaRD-PaWR, \(K_b\) is the number of pages in SrRD-PaWR, and \(K_c\) is the number of pages in SrRD-SrWR \((K_c = K - L)\), then the cost of data movement in PaGC can written as:

\[
\text{Cost}_{\text{PaGC}} = K_a \times (T_{RD} + T_{WR}) + K_b \times (2 \times T_{RD} + T_{WR}) = K_c \times (T_{RD} + T_{WR}),
\]

where \(T_{RD}\) and \(T_{WR}\) are the latencies for reading a page into the cache register and writing it back to the flash.

**B. When does FTL perform PaGC?**

We introduce and evaluate three different strategies at FTL to decide when GC has to be performed in parallel.

- **Blind PaGC** always performs parallel GC. In other words, when GC is invoked for one of the planes, the other plane(s) are also forced to invoke GC. This strategy potentially increases the plane idle time compared to the baseline, because the PaGC latency can be larger than the serial GC.

- **On-demand PaGC** allows parallel GC, if and only if the number of free pages in the paired plane is less than a threshold \((PaGC_{TH})\). This strategy tries to keep the cost of PaGC at a reasonable level. In fact, depending on the mapping function employed by the FTL, the write traffic on different planes in a die may differ considerably; so, it is possible that, when invoking GC on one of the planes, the other plane is far from its GC point of invocation, and we have to avoid its erasure. In this situation, GC is performed on the target plane, similar to the conventional serial GC.

- **On-demand PaGC with cache assistance**, similar to on-demand PaGC, performs parallel GC, if the number of free pages in the paired plane is less than \(PaGC_{TH}\), but reduces the cost PaGC by avoiding the latency of serial write in the above equation. In fact, since a write operation is slow, the cost of PaGC increases significantly if a large number of pages belong to the SrRD-SrWR category. This strategy avoids this cost by reading the pages of this category out of the chip and keep them in the SDRAM buffer on the SSD board (Figure 1). Later, the FTL will commit them in the flash array. As a result, by removing single-plane writes in PaGC, all write-backs are accomplished using the multi-plane write operation. Note that, even after employing this strategy, PaGC may still incur extra cost over the serial GC, due to the serial read operations for the pages in the SrRD-PaWR category. However, this cost is expected to be insignificant as a read takes much lower latency than a write.

**VI. EVALUATION SETUP**

**Simulation platform.** We perform trace-driven simulation using SSDsim [32]. We modified SSDsim to have a detailed implementation of the ONFI model [27], the GC algorithms, and the buffer management algorithms at FCC.

**Target SSD.** We simulated a 800GB enterprise SSDs; detailed in Table I. This SSD has sixteen channels, each with a 333MT/s bandwidth. Each channel is connected to eight flash chips. Each flash chip is a Micron 64Gb (8GB) MLC NAND product [22], also summarized in Table I. We use pure page-level mapping. The allocation scheme is Channel-first, Way-second, Die-third, and Plane-fourth, (CWDP). The FTL triggers the GC process when the ratio of free pages within a plane drops below 7%. We assumed the GREEDY GCselect (its search time is negligible as GC is locally executed within a plane), and the free block selection used is round robin.

**Workloads.** Table II lists the important characteristics of the evaluated workloads, all taken from [34], [35]. Each
TABLE I: Main characteristics of the baseline SSDs.

<table>
<thead>
<tr>
<th>SSD Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>User capacity: 800GB, Spare-factor: 25%, SSD-to-host protocol: PCIe 2.0 x8, 16 Channels, 8 Chips per Channel, Channel rate: 333MT/sec</td>
</tr>
<tr>
<td>NAND flash (Micron [22])</td>
</tr>
<tr>
<td>Page size = 8 KB, Block size = 2MB, Planes per die = 2, Dies per chip = 1, Flash chip capacity = 8 GB, Page read latency = 75µs, Page program latency = 1.5ms, Block erase latency = 3.8ms</td>
</tr>
</tbody>
</table>

TABLE II: Main Characteristics of the evaluated workloads.

<table>
<thead>
<tr>
<th>Trace</th>
<th>WR-RD Ratio [%]</th>
<th>Random WR [%]</th>
<th>Zero-GC RT (%)</th>
</tr>
</thead>
</table>

### Low GC Contribution

<table>
<thead>
<tr>
<th>Trace</th>
<th>WR-RD Ratio [%]</th>
<th>Random WR [%]</th>
<th>Zero-GC RT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fin-1</td>
<td>76.83</td>
<td>70.61</td>
<td>70.65</td>
</tr>
<tr>
<td>proj-0</td>
<td>87.51</td>
<td>17.19</td>
<td>47.0</td>
</tr>
<tr>
<td>prxy-0</td>
<td>96.93</td>
<td>62.58</td>
<td>78.13</td>
</tr>
<tr>
<td>src2-0</td>
<td>88.65</td>
<td>52.21</td>
<td>90.62</td>
</tr>
<tr>
<td>stg-0</td>
<td>84.81</td>
<td>40.59</td>
<td>92.24</td>
</tr>
<tr>
<td>ts-0</td>
<td>82.42</td>
<td>46.75</td>
<td>90.71</td>
</tr>
<tr>
<td>usr-0</td>
<td>59.58</td>
<td>25.65</td>
<td>85.43</td>
</tr>
<tr>
<td>usr-1</td>
<td>8.51</td>
<td>13.79</td>
<td>75.44</td>
</tr>
</tbody>
</table>

### High GC Contribution

<table>
<thead>
<tr>
<th>Trace</th>
<th>WR-RD Ratio [%]</th>
<th>Random WR [%]</th>
<th>Zero-GC RT (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hm-0</td>
<td>64.49</td>
<td>49.41</td>
<td>41.04</td>
</tr>
<tr>
<td>msnfs-0</td>
<td>97.23</td>
<td>27.07</td>
<td>37.30</td>
</tr>
<tr>
<td>msnfs-1</td>
<td>94.52</td>
<td>33.39</td>
<td>25.01</td>
</tr>
<tr>
<td>msnfs-4</td>
<td>23.78</td>
<td>46.12</td>
<td>14.91</td>
</tr>
<tr>
<td>msnfs-5</td>
<td>23.41</td>
<td>44.23</td>
<td>11.33</td>
</tr>
<tr>
<td>pm-0</td>
<td>89.21</td>
<td>41.43</td>
<td>40.52</td>
</tr>
<tr>
<td>proj-2</td>
<td>12.38</td>
<td>15.43</td>
<td>3.98</td>
</tr>
</tbody>
</table>

A workload in this table is characterized by its write-read ratio, write randomness, and zero-GC normalized response time (i.e., response time of a system with zero-latency GC normalized to the baseline system). The last metric in the table gives the maximum improvement that each GC-related optimizations could achieve and used as a performance enhancement cap in our analysis. Our traces are categorized into two groups based on their zero-GC normalized response time: a workload has high GC contribution, if this value is higher than 50%; otherwise, it has low GC contribution.

**Examined systems.** We evaluated four systems:

- **Baseline:** It uses a conventional serial GC.
- **Blind PaGC (B-PaGC):** It always performs PaGC when at least one of the planes in a die invokes GC.
- **Threshold-based PaGC (T-PaGC):** It uses on-demand PaGC. PaGC<sub>T</sub>H is assumed to be 5% larger than the original GC-invocation threshold.
- **Threshold-based PaGC with Cache assistance (TC-PaGC):** On-demand PaGC with cache assistance. PaGC<sub>T</sub>H is set to 5% larger than the original GC-invocation threshold.

### VIII. EXPERIMENTAL RESULTS

We analyze the performance results for the two workload categories separately. For each case, we present performance results in terms of (a) the average response time of all requests normalized to the baseline system; (b) the response time breakdown in terms of service time, latency due to resource conflicts, and various types of conflicts described in Section IV; (c) the breakdown of PaGC execution time in terms of the time it spends for various page movements.

A. Results for the Workloads with Low GC Contribution

1) Impact on response time: Figure 9 plots the request response time of the examined PaGC systems, normalized to the baseline. For each workload, this figure also shows the request response time in a system with zero GC delay, which gives the maximum potential benefits of any GC-related optimization scheme could achieve. One can see from this figure that, when the PaGC is invoked blindly (B-PaGC), the average request response time increases over the baseline in a few workloads (src20, stg0, ts0, usr0). In fact, as B-PaGC always performs parallel GC when one of the planes in a die enters the GC mode, these workloads the number of valid pages in the selected victim block of the non-demanding plane is higher than that of the victim block of the GC-demanding plane. As a result, the delay of GC increases significantly, especially when the non-demanding plane does not need to perform GC in near future (i.e., it has enough free pages or its write traffic is not high to consume its free pages). With the threshold-based PaGC (either with or without cache assistance), this overhead is removed and the FTL runs PaGC when it is really beneficial (i.e., the number of free pages in the non-demanding plane is low). Consequently, for these applications, T-PaGC and TC-PaGC result in 4% to 12% reduction in the request response time, getting close to the zero-delay-GC (in the next subsection, we show that performance of these workloads cannot improve beyond 12% because of the excessive resource conflicts involving non-GC I/Os). For the rest of the workloads, PaGC always reduces the GC cost and improves the request response time between 10% in usr1 to 40% in porj0. We also see that our threshold-based PaGCs (T-PaGC and TC-PaGC) perform always better than the blind PaGC (B-PaGC) due to their cost-aware design.

2) PaGC delay breakdown: Figure 7 plots the average number of page movements in the examined PaGC systems and its breakdown based on various categories (PaRD-PaWR, SrPD-PaWR, SrRD-SrWR for the demanding plane, and SrRD-SrWR for the non-demanding plane). One can see from these results that, for most of the workloads, a small fraction of data movements is accomplished in a completely parallel fashion (PaRD-PaWR), but a large number of them are carried out in the SrRD-PaWR fashion, i.e., still benefiting. It is also clear that for the workloads that B-PaGC degrades the request response time (src20, stg0, ts0, usr0), we have a huge increase in the number of page movements belonging to the SrRD-SrWR(ND) category, which represents the overhead over the original GC delay. T-PaGC and TC-PaGC remove this overhead by executing PaGC more judiciously.

3) Response time breakdown: Figure 8 shows the request response time breakdown. One can see from this plot that...
our PaGC systems (especially threshold-based, T-PaGC and TC-PaGC) always reduce the execution overhead brought by GC (CsGC, CoGC and LC) in all workloads, increasing the contribution of the actual request service time (ST). In fact, the total GC contribution in response time has reduced to less than 10% by the proposed schemes. For these workloads, the non-GC resource conflicts (CnGC) are a great contributor (up to 90%) that cannot be optimized by improving GC alone.

4) Plane utilization: Figure 10 shows average utilization of all planes in the target SSD. One can see that in the Baseline system, the plane has equal time performing GC and waiting for other plane to perform GC. The GC idle time has been removed completely in the B-PaGC approach, since planes are performing GC together all the time, so a plane is never idle waiting for GC in other plane. However, performing PaGC blindly, in some cases (src20", stg0" and usr0") caused performing even more GC than the baseline system. On the other hand, threshold based PaGCs reduce the amount of idle

B. Results for the Workloads with High GC Contribution

1) Impact on response time: Figure 14 plots the response time of the PaGC systems, normalized to the baseline. Again, the zero delay GC gives the maximum improvement achievable by GC optimizing. As expected, due to high GC rate
in these workloads, our PaGC always reduces response time (even when it is performed blindly). In the case of B-PaGC, except for hm0 and prn0, PaGC gives request response time enhancement close to the threshold-based PaGCs (between 40% to 45%). For hm0 and prn0, B-PaGC still improves performance by 15% and 19%, respectively; but, GC performance is further improved when employing T-PaGC and TC-PaGC.

2) PaGC delay breakdown: Figure 11 plots the average number of valid pages moved in different systems for the workloads with high GC contribution. One can see that, in these workloads, a great number of page movements belong to the categories that perform write operations in parallel on two planes (i.e., PaRD-PaWR and SrRD-SrWR). This is the reason why PaGC largely improves request response time for them. The improvement brought by B-PaGC is not as high as the threshold-based PaGCs for hm0 and prn0, since a very large number of page movements are of type SrRD-SrWR for the non-demanding plane. Note that, in porgy, most of the page movements are accomplished in a completely parallel fashion (PaRD-PaWR). The reason is that the number of valid pages in the selected victim blocks is usually high for these workloads, increasing chance for finding aligned pages.

3) Response time breakdown: Figure 12 gives the request response time breakdown for the workloads with high GC contribution. One can see that the resource conflicts (either non-GC or GC-related conflicts) are large in the baseline for these workloads, and PaGC addresses only part of the GC-related ones. The reason for the large amount of GC conflicts after applying PaGC is that the other planes in the same chips execute GC with high rate, keeping the shared resources busy.

4) Plane utilization: Figure 13 shows average utilization of all planes in the target SSD. Again in the Baseline system, the plane has almost equal time performing GC and waiting for other plane to perform GC. B-PaGC approach remove the idle time completely. Threshold based PaGCs reduce the amount of idle time waiting for GC in other plane significantly. In some cases (e.g. hm0 and msnfs1), the extra moving time overhead in PaGC caused increase in the busy time for GC.
PaGC systems evaluated enhances the request response time.

Figure 16 plots the normalized request response time of PaGC.

C. Impact of GCselect Algorithm

Figure 15 plots the impact of different GCselect algorithms on performance efficiency of PaGC. It shows the results for B-PaGC and TC-PaGC (T-PaGC results are very close to those by TC-PaGC) and two GCselect algorithms: RGA and RANDOM. One can see that, similar to the GREEDY algorithm (results in Figures 9–12), B-PaGC degrades the request response time in a couple of workloads with low GC contribution (i.e., src20, stg0, tsp and usr0). However, TC-PaGC is then able to reduce B-PaGC cost and improve performance by up to 7%. For the rest of our workloads, both systems see performance improvements with the RGA algorithm (in range of 6%–35%). For RANDOM GCselect, however, we see performance enhancements in all workloads (except fin1) which is sometimes up to 80% (for prxy0). Hence, irrespective of the GCselect algorithm chosen, PaGC is efficient and reduces response time by 40%, on average.

D. Impact of Page Allocation

Page allocation greatly affects the traffic load on each chip, die and plane [30]–[32] which eventually determines the GC rate and GC latency on each plane. The results in Sections VII-A and VII-B considered channel-first, way-second, die-third, plane-last (CWPD) allocation, which collectively show that PaGC improves the SSD performance by 5%–45%.

Figure 16 plots the normalized request response time of PaGC when page allocation is plane-first, channel-second, way-third and die-last (PCWD). We choose this scheme because it strips sequential page addresses on the adjacent planes in the same die which cause the adjacent planes get nearly the same write traffic and more similarity during GC. This is why that any PaGC systems evaluated enhances the request response time.

E. Comparative Analysis of PaGC and Superblock

In modern flash arrays, superblock [36], [37] is widely used as an approach that creates a cluster of blocks or a cluster pages by first combining blocks in different SSD resources, and then stripping a large-sized request on a block-cluster or page-cluster, thereby maximizing inter-chip and intra-chip parallelism. Assuming a plane-level superblock, the blocks and pages of different planes in the same die are clustered, creating a super-block or super-page spanning on the paired planes. With this plane-level superblock, the FTL uses the super-blocks and super-pages as the granularity for a read or a write. Figure 17 compares the request response time of TC-PaGC and that of super-block scheme, both normalized to the baseline system with configuration in Table I. One can see that, although plane-level superblock increases the plane-level utilization, its efficiency is limited by the access pattern exhibited by the workload being executed (e.g., in msnfs4 and msnfs5, it causes performance degradation by 2.5× and 2.2×, respectively). From this figure, we can also observe that, for some workloads, the plane-level superblock performs better than TC-PaGC (e.g., in msnfs0 and prxy0), while for the other workloads, TC-PaGC gives a better performance.

VIII. Conclusion

This paper attacks one of the least understood problems caused by Garbage Collection (GC) on modern SSDs: plane-level resource underutilization. After discussing the experimental evidence that clearly documents plane-level underutilization, we propose a novel GC strategy called Parallel GC (PaGC) that tries to run concurrent GCs on all planes of a given die whenever one of its planes requires on-demand GC. Our extensive experimental evaluations using various SSD workloads reveal that, while blindly running
PaGc on all planes may not be the best option, constraining its frequency by considering the number of free pages on the planes that are not involved in an on-demand GC can bring significant improvements on response times. More specifically, our results with this proactive GC strategy indicate about 32% (on average) improvement on response time.

ACKNOWLEDGMENT

This work is supported in part by NSF grants 1526750, 1302557, 1213052, 1439021, and 1302225 and NRF grants 2016R1C1B2015312 and 2015M3C4A7065645 and IITP grant 2015-R0346-15-1008 and a grant from Intel.

REFERENCES


2015-R0346-15-1008 and and a grant from Intel.


Micron MT29F64G08CBAAWP NAND Flash Memory datasheet,” http://www.micron.com/parts/nand-flash/mass-storage/mt29f64g08cbaaawp.

Intel JS29F64G08CAMDI MD332 NAND Flash Memory datasheet,” http://intel.com/design.


Open NAND flash interface specification 3.x,” Jun 2013.


Microsoft production server traces,” http://iotta.snia.org/traces/list/BlockIO.
